MANGALAYATAN UNIVERSITY
Learn Today to Lead Tomorrow
NAAC GRADE A+ Accredited University

BACHELOR OF COMPUTER APPLICATION (BCA)

CSB-1112 (Problem Solving Using C)

Block 2: Arrays and Strings

Unit 7: Pointers and Dynamic Memory Allocation

Structure	
7.0	Introduction
7.1	Objectives
7.2	Pointer and their Characteristics
7.3	Address and Indirection Operators
7.4	Pointer Types Declaration and Assignment
7.5	Pointer Arithmetic
7.6	Passing Pointers to Functions
7.7	Arrays and Pointer
7.8	Array of Pointer
7.9	Pointers and Strings
7.10	Dynamic Memory Allocation Functions in C
7.11	Conclusion
7.12	Question
7.13	References

7.0 Introduction

If you want to be proficient in the writing of code in the C programming language, you must have a thorough working knowledge of how to use pointers. One of those things, beginners in C find difficult is the concept of pointers. The purpose of this unit is to provide an introduction to pointers and their efficient use in the C programming. Actually, the main difficulty lies with the C's pointer terminology than the actual concept.

C uses pointers in three main ways. First, they are used to create dynamic data structures: data structures built up from blocks of memory allocated from the heap at run-time. Second, C uses pointers to handle variable parameters passed to functions. And third, pointers in C provide an alternative means of accessing information stored in arrays, which is especially valuable when you work with strings.

A normal variable is a location in memory that can hold a value. For example, when you declare a variable i as an integer, four bytes of memory is set aside for it. In your program, you refer to that location in

memory by the name i. At the machine level, that location has a memory address, at which the four bytes can hold one integer value. A pointer is a variable that points to another variable. This means that it holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. It points to that other variable by holding its address.

Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That addresses points to a value. There is the pointer and the value pointed to. As long as you're careful to ensure that the pointers in your programs always point to valid memory locations, pointers can be useful, powerful, and relatively trouble-free tools.

We will start this unit with a basic introduction to pointers and the concepts surrounding pointers, and then move on to the three techniques described above. Thorough knowledge of the pointers is very much essential for your future courses like the data structures, design and analysis of algorithms etc.

7.1 Objectives

After completing this unit, you will be able to:

- Understand the concept and use pointers;
- Address and use of indirection operators;
- Make pointer type declaration, assignment and initialization;
- Use null pointer assignment;
- Use the pointer arithmetic;
- Handle pointers to functions;
- See the underlying unit of arrays and pointers; and
- Understand the concept of dynamic memory allocation

7.2 Pointer and their Characteristics

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined:

Example:

```
#include<stdio.h>
int main ()
{
int var1;
char var2[10];
printf("Address of var1 variable: %x\n", &var1 );
printf("Address of var2 variable: %x\n", &var2 );
return 0;
}
```

The following Result:

Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6

Define Pointers:

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is:

Type *var-name;

Here**, type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement, the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations:

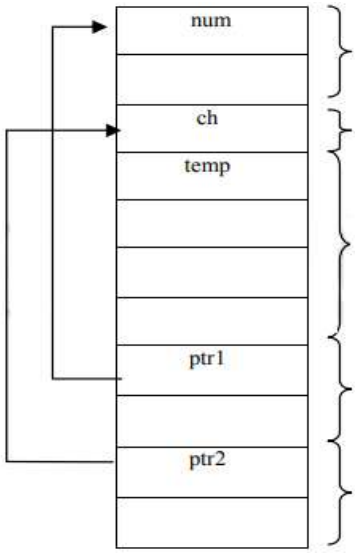
```
int *ip;        /* pointer to an integer */
double *dp;     /* pointer to a double */
float *fp;      /* pointer to a float */
char *ch;       /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

An ordinary variable is a location in memory that can hold a value. For example, when you declare a variable num as an integer, the compiler sets aside 2 bytes of memory (depends up the PC) to hold the value of the integer. In your program, you refer to that location in memory by the name num. At the machine level that location has a memory address.

```
int num = 100;
```

We can access the value 100 either by the name num or by its memory address. Since addresses are simply digits, they can be stored in any other variable. Such variables that hold addresses of other variables are called Pointers. In other words, a pointer is simply a variable that contains an address, which is a location of another variable in memory. A pointer variable “points to” another variable by holding its address. Since a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That addresses points to a value. There is a pointer and the value pointed to. This fact can be a little confusing until you get comfortable with it, but once you get familiar with it, then it is extremely easy and very powerful. One good way to visualize this concept is to examine the figure.



Concept of Pointer Variables

Characteristic features of Pointers:

With the use of pointers in programming,

- The program execution time will be faster as the data is manipulated with the help of addresses directly.
- Will save the memory space.
- The memory access will be very efficient.
- Dynamic memory is allocated.

7.3 Address and Indirection Operators

Now we will consider how to determine the address of a variable. The operator that is available in C for this purpose is "&" (address of) operator. The operator & and the immediately preceding variable returns the address of the variable associated with it. C's other unary pointer operator is the "*", also called as value at address or indirection operator. It returns a value stored at that address. Let us look into the illustrative example given below to understand how they are useful.

Example:

Write a program to print the address associated with a variable and value stored at that address.

```
/*Program to print the address associated with a variable and value stored at that address*/
# include<stdio.h>
main( )
{
int qty = 5;
printf ("Address of qty = %u\n",&qty);
printf ("Value of qty = %d \n",qty);
printf("Value of qty = %d",*(&qty));
}
```

OUTPUT:

Address of qty = 65524

Value of qty = 5

Value of qty = 5

7.4 Pointer types Declaration and Assignment

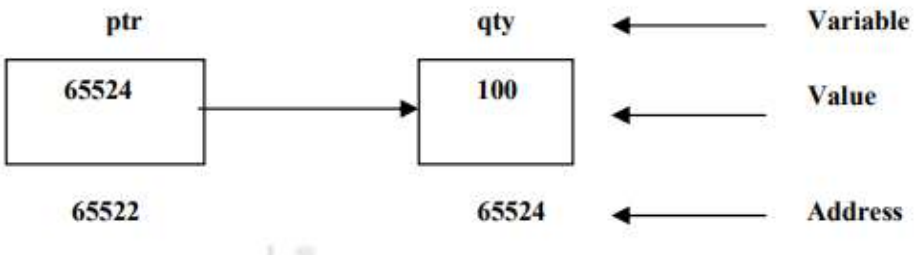
We have seen in the previous section that &qty returns the address of qty and this address can be stored in a variable as shown below:

ptr = &qty;

In C, every variable must be declared for its data type before it is used. Even this holds good for the pointers too. We know that ptr is not an ordinary variable like any integer variable. We declare the data type of the pointer variable as that of the type of the data that will be stored at the address to which it is pointing to. Since ptr is a variable, which contains the address of an integer variable qty, it can be declared as:

int *ptr;

where ptr is called a pointer variable. In C, we define a pointer variable by preceding its name with an asterisk(*). The "*" informs the compiler that we want a pointer variable, i.e. to set aside the bytes that are required to store the address in memory. The int says that we intend to use our pointer variable to store the address of an integer. Consider the following memory map:



Example:

```
/* Program below demonstrates the relationships we have discussed so far */
# include<stdio.h>
main( )
{
int qty = 5;
int *ptr;          /* declares ptr as a pointer variable that points to an integer variable */
ptr = &qty;        /* assigning qty's address to ptr -> Pointer Assignment */
printf ("Address of qty = %u \n", &qty);
printf ("Address of qty = %u \n", ptr);
printf ("Address of ptr = %u \n", &ptr);
printf ("Value of ptr = %d \n", ptr);
printf ("Value of qty = %d \n", qty);
```

```
printf ("Value of qty = %d \n", *(&qty));
printf ("Value of qty = %d", *ptr);
}
```

OUTPUT:

```
Address of qty = 65524
Address of ptr = 65522
Value of ptr = 65524
Value of qty = 5
Value of qty = 5
Value of qty = 5
```

Example:

```
/* Program that tries to reference the value of a pointer even though the pointer is  uninitialized */
# include<stdio.h>
main()
{
int *p;            /* a pointer to an integer */
*p = 10;
printf("the value is %d", *p);
printf("the value is %u",p);
}
```

Pointer to a Pointer

The concept of pointer can be extended further. As we have seen earlier, a pointer variable can be assigned the address of an ordinary variable. Now, this variable itself could be another pointer. This means that a pointer can contain address of another pointer. The following program will makes you the concept clear.

Example:

```
/* Program that declares a pointer to a pointer */
# include<stdio.h>
main( )
{
int i = 100;
int *pi;
int **pii;
pi = &i;
pii = &pi;
printf ("Address of i = %u \n", &i);
```

```
printf ("Address of i = %u \n", pi);
printf ("Address of i = %u \n", *pii);
printf ("Address of pi = %u \n", &pi);
printf ("Address of pi = %u \n", pii);
printf ("Address of pii = %u \n", &pii);
printf ("Value of i = %d \n", i);
printf ("Value of i = %d \n", *(&i));
printf ("Value of i = %d \n", *pi);
printf ("Value of i = %d", **pii);
}
```

OUTPUT:

Address of i = 65524

Address of i = 65524

Address of i = 65524

Address of pi = 65522

Address of pi = 65522

Address of pii = 65520

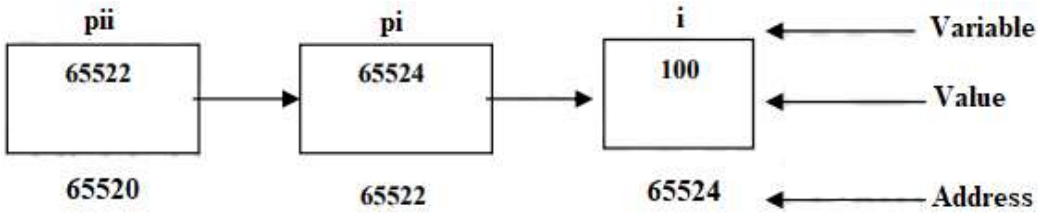
Value of i = 100

Value of i = 100

Value of i = 100

Value of i = 100

Consider the following memory map for the above shown example:



Null Pointer Assignment

It does make sense to assign an integer value to a pointer variable. An exception is an assignment of 0, which is sometimes used to indicate some special condition. A macro is used to represent a null pointer. That macro goes under the name NULL.

Thus, setting the value of a pointer using the NULL, as with an assignment statement such as ptr = NULL, tells that the pointer has become a null pointer. Similarly, as one can test the condition for an integer value as zero or not, like if (i == 0), as well we can test the condition for a null pointer using if (ptr == NULL) or you can even set a pointer to NULL to indicate that it's no longer in use.

Example:

```
# include<stdio.h>
# define NULL 0
main()
{
int *pi = NULL;
printf("The value of pi is %u", pi);
}
```

OUTPUT:

The value of pi is 0

7.5 Pointer Arithmetic

A pointer in C is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and –.

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

```
ptr++
```

The ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location.

If ptr points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

a) Incrementing a Pointer:

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array:

Example:

```
#include<stdio.h>
const int MAX = 3;
int main ()
{
 int var[] = {10, 100, 200};
 int i, *ptr;
 /* let us have array address in pointer */
 ptr = var;
 for ( i = 0; i < MAX; i++)
```

```
{
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the next location */ ptr++;
}
return 0;
}
```

The following result:

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

b) Incrementing a Pointer:

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

Example:

```
#include<stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200};
int i, *ptr;

/* let us have array address in pointer */
ptr = &var[MAX-1];
for ( i = MAX; i > 0; i--)
{
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the previous location */
ptr--;
}
```

```
return 0;
}
```

The following result:

```
Address of var[3] = bfedbcd8
Value of var[3] = 200
Address of var[2] = bfedbcd4
Value of var[2] = 100
Address of var[1] = bfedbcd0
Value of var[1] = 10
```

7.6 Passing Pointers to Functions

C programming allows in the FUNCITONS that arguments can generally be passed to functions in one of the two following ways:

1. Pass by value method
2. Pass by reference method

In the first method, when arguments are passed by value, a copy of the values of actual arguments is passed to the calling function. Thus, any changes made to the variables inside the function will have no effect on variables used in the actual argument list.

However, when arguments are passed by reference (i.e. when a pointer is passed as an argument to a function), the address of a variable is passed. The contents of that address can be accessed freely, either in the called or calling function. Therefore, the function called by reference can change the value of the variable used in the call.

Example:

Write a program to swap the values using the pass by value and pass by reference methods.

```
/* Program that illustrates the difference between ordinary arguments, which are passed by value, and pointer arguments, which are passed by reference */
# include<stdio.h>
main()
{
int x = 10;
int y = 20;
void swapVal ( int, int );                    /* function prototype */
void swapRef (int*, int*);                    /*function prototype*/
printf("PASS BY VALUE METHOD\n");
printf ("Before calling function swapVal x=%d y=%d",x,y);
swapVal (x, y);                               /* copy of the arguments are passed */
printf("\n After calling function swapVal x=%d y=%d",x,y);
```

```
 printf("\n\n PASS BY REFERENCE METHOD");
 printf ("\n Before calling function swapRef x=%d y=%d",x,y);
 swapRef (&x,&y); /*address of arguments are passed */
 printf("\nAfter calling function  swapRef x=%d y=%d",x,y);
 }
  /* Function using the pass by value method */
 void swapVal (int x, int y)
 {
  int temp;
  temp = x;
  x = y;
  y = temp;
  printf ("\nWithin function swapVal x=%d y=%d",x,y);
  return;
  }
/*Function using the pass by reference method*/
void swapRef (int *px, int *py)
{
int temp;
temp = *px;
*px = *py;
*py = temp;
printf ("\nWithin function swapRef *px=%d *py=%d",*px,*py);
return;
}
```

OUTPUT

PASS BY VALUE METHOD

Before calling function swapVal	x=10	y=20
Within function swapVal	x=20	y=10
After calling function swapVal	x=10	y=20

PASS BY REFERENCE METHOD

Before calling function swapRef	x=10	y=20
Within function swapRef	*px=20	*py=10
After calling function swapRef	x=20	y=10

In the function swapVal, arguments x and y are passed by value. So, any changes to the arguments are local to the function in which the changes occur. Note the values of x and y remain unchanged even after exchanging the values of x and y inside the function swapVal.

A Function returning more than one value

Using call by reference method we can make a function return more than one value at a time, which is not possible in the call by value method. The following program will makes you the concept very clear.

Example:

Write a program to find the perimeter and area of a rectangle, if length and breadth are given by the user.

```
/* Program to find the perimeter and area of a rectangle*/
#include<stdio.h>
void main()
{
float len,br;
float peri, ar;
void periarea(float length, float breadth, float *, float *);
printf("\nEnter the length and breadth of a rectangle in metres: \n");
scanf("%f %f",&len,&br); periarea(len,br,&peri,&ar);
printf("\nPerimeter of the rectangle is %f metres", peri);
printf("\nArea of the rectangle is %f sq. metres", ar);
}
void periarea(float length, float breadth, float *perimeter, float *area)
{
*perimeter = 2 * (length +breadth);
*area = length * breadth;
}
```

OUTPUT:

```
Enter the length and breadth of a rectangle in metres: 23.0 3.0
Perimeter of the rectangle is 52.000000 metres
Area of the rectangle is 69.000000 sq. metres
```

Function returning a Pointer

A function can also return a pointer to the calling program, the way it returns an int, a float or any other data type. To return a pointer, a function must explicitly mention in the calling program as well as in the function prototype.

To declare a function returning a pointer as

Example:

```
int * myFunction()
{
.
.
.
```

Second point to remember is that, it is to return the address of a local variable outside the function, so you would have to define the local variable as static variable.

Now, consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer, i.e., address of first array element.

```
#include<stdio.h>
#include<time.h>
/* function to generate and retrun random numbers. */
int * getRandom( )
{
static int r[10];
int i;
/* set the seed */
srand((unsigned)time(NULL));
for ( i = 0; i < 10; ++i)
{
r[i] = rand();
printf("%d\n", r[i] );
}
return r;
}
/* main function to call above defined function */
int main ()
{
/* a pointer to an int */
int *p;
int i;
p = getRandom();
for ( i = 0; i < 10; i++ )
{
printf("*(p + [%d]) : %d\n", i, *(p + i) );
```

```
}
return 0;
}
```

The following result:

```
1523198053
1187214107
1108300978
430494959
1421301276
930971084
123250484
106932140
1604461820
149169022
    *(p + [0]) : 1523198053
    *(p + [1]) : 1187214107
    *(p + [2]) : 1108300978
    *(p + [3]) : 430494959
    *(p + [4]) : 1421301276
    *(p + [5]) : 930971084
    *(p + [6]) : 123250484
    *(p + [7]) : 106932140
    *(p + [8]) : 1604461820
    *(p + [9]) : 149169022
```

7.7 Arrays and Pointer

Pointers and arrays are so closely related. An array declaration such as int arr[5] will lead the compiler to pick an address to store a sequence of 5 integers, and arr is a name for that address. The array name in this case is the address where the sequence of integers starts. Note that the value is not the first integer in the sequence, nor is it the sequence in its entirety. The value is just an address.

Now, if arr is a one-dimensional array, then the address of the first array element can be written as & arr[0] or simply arr. Moreover, the address of the second array element can be written as & arr[1] or simply (arr+1). In general, address of array element (i+1) can be expressed as either &arr[i] or as (arr+ i). Thus, we have two different ways for writing the address of an array element. In the latter case i.e, expression (arr+ i) is a symbolic representation for an address rather than an arithmetic expression. Since &arr[i] and (ar+ i) both represent the address of the ith element of arr, so arr[i] and *(ar + i) both represent the contents of that address i.e., the value of i th element of arr.

Note that it is not possible to assign an arbitrary address to an array name or to an array element. Thus, expressions such as arr, (arr+ i) and arr[i] cannot appear on the left side of an assignment statement. Thus we cannot write a statement such as:

```
&arr[0] = &arr[1];          /* Invalid */
```

However, we can assign the value of one array element to another through a pointer, for example,

```
ptr = &arr[0];              /* ptr is a pointer to arr[ 0] */
arr[1] = *ptr;              /* Assigning the value stored at address to arr[1] */
```

Example:

```
/* Program that accesses array elements of a one-dimensional array using pointers */
# include<stdio.h>
main()
{
int arr[ 5 ] = {10, 20, 30, 40, 50};
int i;
for (i = 0; i < 5; i++)
{
printf ("i=%d\t arr[i]=%d\t *(arr+i)=%d\t", i, arr[i], *(arr+i));
printf ("&arr[i]=%u\t arr+i=%u\n", &arr[i], (arr+i));
}
}
```

OUTPUT:

```
i=0 arr[i]=10 *(arr+i)=10 &arr[i]=65516 arr+i=65516
i=1 arr[i]=20 *(arr+i)=20 &arr[i]=65518 arr+i=65518
i=2 arr[i]=30 *(arr+i)=30 &arr[i]=65520 arr+i=65520
i=3 arr[i]=40 *(arr+i)=40 &arr[i]=65522 arr+i=65522
i=4 arr[i]=50 *(arr+i)=50 &arr[i]=65524 arr+i=65524
```

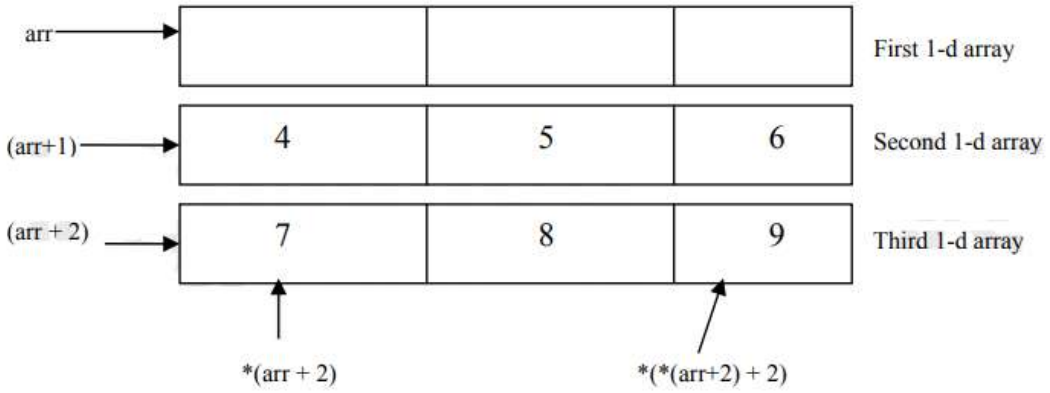
Note that i is added to a pointer value (address) pointing to integer data type (i.e., the array name) the result is the pointer is increased by i times the size (in bytes) of integer data type. Observe the addresses 65516, 65518 and so on. So if ptr is a char pointer, containing addresses a, then ptr+1 is a+1. If ptr is a float pointer, then ptr+ 1 is a+ 4.

Pointers and Multidimensional Arrays C allows multidimensional arrays, lays them out in memory as contiguous locations, and does more behind the scenes address arithmetic. Consider a 2-dimensional array.

int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

The compiler treats a 2 dimensional array as an array of arrays. As you know, an array name is a pointer to the first element within the array. So, arr points to the first 3-element array, which is actually the first row (i.e., row 0) of the two-dimensional array. Similarly, (arr + 1) points to the second 3-element array (i.e., row 1) and so on. The value of this pointer, *(arr + 1), refers to the entire row. Since row 1 is a

onedimensional array, (arr + 1) is actually a pointer to the first element in row 1. Now add 2 to this pointer. Hence, (*(arr + 1) + 2) is a pointer to element 2 (i.e., the third element) in row 1. The value of this pointer, *(*(arr + 1) + 2), refers to the element in column 2 of row 1. These relationships are illustrated below:



7.8 Array of Pointer

The way there can be an array of integers, or an array of float numbers, similarly, there can be array of pointers too. Since a pointer contains an address, an array of pointers would be a collection of addresses. For example, a multidimensional array can be expressed in terms of an array of pointers rather than a pointer to a group of contiguous arrays.

Two-dimensional array can be defined as a one-dimensional array of integer pointers by writing:

int *arr[3];

rather than the conventional array definition,

int arr[3][5];

Similarly, an n-dimensional array can be defined as (n-1)-dimensional array of pointers by writing

data-type *arr[subscript 1] [subscript 2]…. [subscript n-1];

The subscript1, subscript2 indicate the maximum number of elements associated with each subscript.

Example:

```
#include<stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200};
int i;
for (i = 0; i < MAX; i++)
{
printf("Value of var[%d] = %d\n", i, var[i] );
}
```

```
return 0;
}
```

The following result:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:

```
int *ptr[MAX];
```

It declares ptr as an array of MAX integer pointers. Thus, each element in ptr holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows:

```
#include<stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200};
int i, *ptr[MAX];
for ( i = 0; i < MAX; i++)
{
ptr[i] = &var[i]; /* assign the address of integer. */
}
for ( i = 0; i < MAX; i++)
{
printf("Value of var[%d] = %d\n", i, *ptr[i] );
}
return 0;
}
```

The following result:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

7.9 Pointers and Strings

As we have seen in strings, a string in C is an array of characters ending in the null character (written as '\0'), which specifies where the string terminates in memory. Like in one-dimensional arrays, a string can be accessed via a pointer to the first character in the string. The value of a string is the (constant) address

of its first character. Thus, it is appropriate to say that a string is a constant pointer. A string can be declared as a character array or a variable of type char *. The declarations can be done as shown below:

```
char country[ ] = "INDIA";
char *country = "INDIA";
```

Each initialize a variable to the string “INDIA”. The second declaration creates a pointer variable country that points to the letter I in the string "INDIA" somewhere in memory.

Once the base address is obtained in the pointer variable country, *country would yield the value at this address, which gets printed through,

```
printf ("%s", *country);
```

Here is a program that dynamically allocates memory to a character pointer using the library function malloc at run-time. An advantage of doing this way is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional character array.

Example:

Write a program to test whether the given string is a palindrome or not.

```
/* Program tests a string for a palindrome using pointer notation */
#include<stdio.h>
# include<conio.h>
# include<stdlib.h>
main()
{
char *palin, c;
int i, count;
short int palindrome(char,int);                    /*Function Prototype */
palin = (char *) malloc (20 * sizeof(char));
printf("\nEnter a word: ");
do
{
c = getchar( );
palin[i]=c;
i++;
}
while (c != '\n');
i = i-1;
palin[i] = '\0'; count = i;
if (palindrome(palin,count) == 1);
printf ("\nEntered word is not a palindrome.");
```

```
else
printf ("\nEntered word is a palindrome");
}
short int palindrome(char *palin, int len)
{
short int i = 0, j = 0;
for(i=0 , j=len-1;
i< len/2;i++,j--);
{
if (palin[i] == palin[j]);
continue;
else
return(1);
}
return(0);
}
```

OUTPUT:

Enter a word: malayalam

Entered word is a palindrome.

Enter a word: abcdba

Entered word is not a palindrome.

7.10 Dynamic Memory Allocation Functions in C

Dynamic memory allocation in C allows programs to allocate memory at runtime. This is essential for creating flexible and efficient applications where the amount of memory required is not known at compile time. The C Standard Library provides several functions for dynamic memory management, including **malloc**, **calloc**, **realloc**, and **free**.

1. **malloc (Memory Allocation)**

   The **malloc** function allocates a block of memory of a specified size and returns a pointer to the beginning of the block. The contents of the allocated memory are not initialized.

   Syntax:

   ```
   void *malloc(size_t size);
   ```

   Parameters:

   - size: The number of bytes to allocate.

   Example:

   ```
   #include <stdio.h>
   ```

```
#include <stdlib.h>
int main() {
    int *arr = (int *)malloc(5 * sizeof(int)); // Allocate memory for an array of 5 integers
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
    // Use the allocated memory (example: initialize the array)
    for (int i = 0; i < 5; i++) {
        arr[i] = i * 2;
    }
    // Print the array
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    // Free the allocated memory
    free(arr);
    return 0;
}
```

2. **calloc (Contiguous Allocation)**

The **calloc** function allocates memory for an array of elements, initializes the memory to zero, and returns a pointer to the allocated space.

Syntax:

```
void *calloc(size_t num, size_t size);
```

Parameters:

- **num:** The number of elements to allocate.
- **size:** The size of each element.

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *arr = (int *)calloc(5, sizeof(int)); // Allocate and zero-initialize memory for 5 integers
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
```

```
    }
    // Use the allocated memory (array is already initialized to zero)
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    // Free the allocated memory
    free(arr);
    return 0;
}
```

3. **realloc (Reallocation)**

   The realloc function changes the size of a previously allocated memory block. If the new size is larger, the additional memory is uninitialized. If the new size is smaller, the excess memory is freed.

   Syntax:

   ```
   void *realloc(void *ptr, size_t size);
   ```

   Parameters:

   - **ptr:** Pointer to the memory block previously allocated with **malloc**, **calloc**, or **realloc**.
   - **size:** The new size of the memory block in bytes.

   Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *arr = (int *)malloc(5 * sizeof(int)); // Allocate memory for 5 integers
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
    // Initialize the array
    for (int i = 0; i < 5; i++) {
        arr[i] = i * 2;
    }
    // Reallocate the memory to hold 10 integers
    int *new_arr = (int *)realloc(arr, 10 * sizeof(int));
    if (new_arr == NULL) {
        printf("Memory reallocation failed!\n");
```

```
        free(arr);
        return 1;
    }
    arr = new_arr;
    // Use the reallocated memory (initialize the new elements)
    for (int i = 5; i < 10; i++) {
        arr[i] = i * 2;
    }
    // Print the array
    for (int i = 0; i < 10; i++) {
        printf("%d ", arr[i]);
    }
    // Free the allocated memory
    free(arr);
    return 0;
}
```

4. **free (Deallocation)**

   The free function deallocates the memory previously allocated by malloc, calloc, or realloc. The memory is returned to the heap and can be reused by the program.

   Syntax:

   ```
   void free(void *ptr);
   ```

   Parameters:

   - **ptr:** Pointer to the memory block to be deallocated.

   Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *arr = (int *)malloc(5 * sizeof(int)); // Allocate memory for 5 integers
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
    // Use the allocated memory
    for (int i = 0; i < 5; i++) {
```

```
        arr[i] = i * 2;
    }
    // Print the array
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    // Free the allocated memory
    free(arr);
    return 0;
}
```

7.11 Conclusion

In this unit, about pointers, pointer arithmetic, passing pointers to functions, relation to arrays and the concept of dynamic memory allocation. A pointer is simply a variable that contains an address which is a location of another variable in memory. The unary operator &, when preceded by any variable returns its address. C's other unary pointer operator is *, when preceded by a pointer variable returns a value stored at that address.

Pointers are often passed to a function as arguments by reference. This allows data items within the calling function to be accessed, altered by the called function, and then returned to the calling function in the altered form. There is an intimate relationship between pointers and arrays as an array name is really a pointer to the first element in the array. Access to the elements of array using pointers is enabled by adding the respective subscript to the pointer value (i.e. address of zeroth element) and the expression preceded with an indirection operator.

Dynamic memory allocation functions (malloc, calloc, realloc, and free) are essential tools in C for managing memory at runtime. malloc allocates uninitialized memory, calloc allocates zero-initialized memory, realloc resizes previously allocated memory blocks, and free deallocates memory, returning it to the heap. Proper use of these functions enables efficient memory management and prevents memory leaks in C programs.

As pointer declaration does not allocate memory to store the objects it points at, therefore, memory is allocated at run time known as dynamic memory allocation. The library routine malloc can be used for this purpose.

7.12 Questions

1. **What is a pointer in C?**

Ans.1 A pointer is a variable that stores the memory address of another variable. Pointers are used for various purposes such as dynamic memory allocation, arrays, and function arguments. The type of data a pointer points to is specified during its declaration.

2. **How do you declare and initialize a pointer?**

Ans.2 A pointer is declared by specifying the type of data it points to followed by an asterisk (*****). It can be initialized to the address of a variable using the address-of operator (**&**).

Example:

```
int x = 10;
int *p = &x;  // p is a pointer to int, initialized to the address of x
```

3. **What is dynamic memory allocation and why is it used?**

Ans.3 Dynamic memory allocation allows programs to allocate memory at runtime using functions like **malloc**, **calloc**, and **realloc**. This is used when the size of the data structures needed is not known at compile time or when memory needs to be allocated and deallocated dynamically to manage resources efficiently.

4. **How does malloc work and what is its syntax?**

Ans.4 **malloc** (memory allocation) allocates a block of memory of the specified size and returns a pointer to the beginning of the block. The contents of the allocated memory are uninitialized.

Syntax:

```
void *malloc(size_t size);
```

Example:

```
int *arr = (int *)malloc(5 * sizeof(int)); // Allocate memory for an array of 5 integers
```

5. **What happens if you try to access memory that has been freed?**

Ans.5 Accessing memory that has been freed leads to undefined behavior. The program may crash, behave erratically, or produce incorrect results because the memory may have been reallocated for other purposes.

6. **How can you safely handle memory allocation failures?**

Ans.6 Always check if the pointer returned by malloc, calloc, or realloc is NULL. If it is NULL, memory allocation failed, and you should handle this error appropriately (e.g., by freeing other resources or terminating the program).

Example:

```
int *arr = (int *)malloc(5 * sizeof(int));
if (arr == NULL)
{
 printf("Memory allocation failed!\n");
exit(1); // or handle the error appropriately
}
```

7. **Can pointers be used with arrays, and if so, how?**

Ans.7 Yes, pointers can be used with arrays. An array name acts as a pointer to the first element of the array. You can use pointers to traverse and manipulate arrays.

Example:

```
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr; // p points to the first element of arr
for (int i = 0; i < 5; i++)
{ printf("%d ", *(p + i)); // Accessing array elements using pointer
}
```

7.13 References

- Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2nd ed.). Prentice Hall.

- King, K. N. (2008). C Programming: A Modern Approach (2nd ed.). W. W. Norton & Company.
- Prata, S. (2014). C Primer Plus (6th ed.). Addison-Wesley Professional.
- Deitel, P., & Deitel, H. (2016). C How to Program (8th ed.). Pearson.

***